Colliding Pucks Solved Using a Temporal Logic

John G. Cleary
Jade Simulations International Corporation and
The University of Calgary

Abstract

A Horn Clause logic programming language, called Starlog, which allows execution of programs involving time is described. A sound and complete bottom up execution procedure for the language is described. An extended example of Programming in Starlog is given in the form of a solution to the colliding pucks problem. A discussion of the features necessary for a distributed implementation of Starlog are given.

Introduction

There have been a number of attempts to use Logic Programming for simulation and in particular to express simulation in a way that could be effectively parallelised. A good survey of recent work on Temporal Logic is given in [Galton.1987]. These logics are based around the idea of new modal logical operators such as next, and until. Unfortunately they seem to be very difficult to use in a simulation context. Also many of them consider time to be a series of discrete instants rather than as a real number.

Logic programming has been an area of particular interest in the attempt to devise effective distributed computing schemes. The primary reason for this has been that the order of execution of parts of a Prolog program do not affect the correctness of the answers to the program. It seems then that this should allow the parts of the program to be executed independently of each other on different processors. Unfortunately this goal has been more elusive than at first anticipated. The committed-choice languages (CP, Parlog etc.) exploit the AND-parallelism of programs. That is the fact that individual clauses within a Prolog goal can be executed in parallel. The main difficulty here is managing the binding of variables which are shared between the different parts of the system as well as the problems of correct semantics mentioned above. Other attempts at exploiting and-parallelism rely on a static analysis of a program to work out what parallelism is available. The main problem here is the difficulty of the analysis and the fact that much potential parallelism can be missed. Or-parallel schemes exploit the fact that different clauses within a program can be executed in any order. The main difficulty with these systems is the difficulty of maintaining many different binding environments in parallel and of managing the space required by the alternatives. A number of attempts have been made to exploit both AND and OR parallelism simultaneously.

Starlog is closely related to the Logic Data Language (LDL) which has been developed as a way of exploiting AND and OR parallelism in data-base applications. LDL uses a forward deduction technique where deduction proceeds from the facts in the data-base toward the final conclusions sought in the query to the data-base. The computation technique used is similar in its details to the relational algebra, with analogs of join, project and other operators. Iterative fixed point computations are used to deal with recursive rules.

This paper introduces another attempt at including simulation within the realms of Logic Programming. The major advantage of this approach is that it has a precise and simple minimal model semantics. The language, called Starlog, retains the standard Horn-clause form of pure Prolog although the resultant programs are interpreted and executed somewhat differently. It is not possible as in standard Prolog to use don't-know non-determinism to search for answers. While don't-know non-determinism is a powerful programming tool it introduces the problem that non-terminating programs (or at least very long running ones) cannot be used because the stack size grows linearly with the execution time. Committed choice Prologs avoid this at the same cost as Starlog of not being able to search for solutions and as well lose a simple semantics.

This paper first introduces the general constructs of Starlog and then uses an extended example of a simulation problem to show the power and expressiveness of the language and to explain its execution mechanism. No attempt is made in this paper to give a formal rendition of its semantics or of an execution mechanism. After this the execution mechanism is explained in more detail and techniques for running it on a multi-computer are described.

Starlog is applicable to a number of domains besides simulation. For example, it includes relational temporal databases as a subset of the language in the same way that relational databases are a subset of pure Prolog. It is also suitable for general purpose programming and systems programming. This paper does not explore these possibilities.

Starlog

Tuples

The fundamental data object in Starlog is a *tuple* (this term is used in analogy with tuples in relational databases and in the same sense that it has been used in Linda [Carriero and Gelernter,1989]). A tuple is a *literal* (see [Lloyd, 1987] for the logic programming terminology used here) which is true over some period of time. The first argument position in each tuple is reserved for a time parameter. For example,

```
        p([2,3),f(g(a)),b)
```

says that `p(T,f(g(a)),b)` is true for all values of $T,\ 2 \leq T < 3$. A tuple can be true over an interval of time or at a single instant as in:

```
        p([5.5,5.5],a,a)
```

which says that p(T,a,a) is true at T = 5.5. Intervals do not appear explicitly anywhere in Starlog, only assertions and results concerning instants of time. For example the tuple above could be generated by the following clause:

```
        p(T,f(g(a)),b) ← 2≤T, T<3.
```

Thus the interval [2,3) above can be taken as shorthand for an infinite number of time instants. To distinguish the standard mathematical usage of brackets for representing open and closed intervals from the standard logic programming usage of brackets for delimiting lists and parameters I have put the mathematical brackets in bold face.

The result of a Starlog program is a (possibly infinite) set of tuples. The set of tuples is all possible answers generated by the clauses in the program, so no query need be supplied. This is similar to the database interpretation of pure Prolog where the result of a query is the whole set of resulting tuples. It is different from the standard Prolog approach where each answer is a separate binding of variables in a query and each of these answers is seen as being an alternative answer to the original query rather than part of one overall answer. It is this change of viewpoint which results in the inability of Starlog to search for alternative answers as it would have to produce entire sets of alternative tuples. Starlog programs compute using forward deduction. The execution is forward in two senses; it proceeds forward from tuples which are known to be true to derive more true tuples and it advances forward thru time in the same way as standard event list driven simulators.

Programs

The simplest form of Starlog program is a unit clause which states that a particular tuple is true. For example

```
p(1.5,a,b) ←
```

says that `p(T,a,b)` is true at the instant `T=1.5`. Every program must contain one or more such unit clauses to start the forward reasoning process. To assert that a tuple is true over an interval of time constraints need to be placed on the time parameter. So

```
p(T,c,d) ← T>1.5, T≤2.3.
```

unconditionally generates the tuple `p((1.5,2.3],c,d)`.

To create more interesting programs it is necessary to allow more general conditions in the body of a clause. For example to say that a tuple `p` will be true `1.5` time units after a tuple `q`, the following clause can be used:

```
p(T0) ← q(T1), T0 == T1 + 1.5.
```

(== is used here to represent arithmetic equality). Execution of such programs starts by generating all tuples from unit clauses and placing them in a *"tuple pool"*. As each tuple is placed in the pool all clauses which have a matching predicate in their body fire and generate a new tuple. For example if the tuple `q(2)` were to appear in the pool then the rule above would fire and generate the tuple `p(3.5)`.

The following is a very simple Starlog program using these principles. It generates a sequence of tuples `integer(T)` true for `T=0, T=1, ....`:

```
integer(0) ←
integer(T0) ← integer(T1), T0 == T1+1.
```

To be more precise the body of a Starlog clause can contain zero or more literals which potentially match tuples as well as built in calls such as <, ≤, and ==. To be effectively computable the clauses must obey two restrictions:

- the first parameter of each tuple must be a time value. That is, it must be a real number ≥ 0.
- the time in the head of the clause must be greater than or equal to the times of all tuples in the body. This is a causality principle, which says that the consequences of any tuple must lie in its future.

I will place these conditions explicitly in clauses in the following examples whenever it is necessary (in a practical programming system they would be added automatically).

Using just what we have seen of the language it is possible to write some simple programs. One typical example is the following which generates a continuous stream of customers (this might be part of a larger queueing simulation). Each customer is generated by a single tuple, `customer(T,Id,R)` where `T` is the time of the customers arrival, `Id` is a unique integer identifier allocated to the customer and `R` is a random number seed used to generate the next customer. The following program consists of two clauses. The first unit clause specifies the first customer who starts the entire process. The second clause generates a new customer a suitable time after the generation of the preceding one.

```
customer(0,1,123456789) ←           %First customer at time 0
%Generate new customer appropriate time after the last one
customer(T0,N,NewSeed) ←                 %Create the new customer
tuple
      customer(T1,q(Id,M),Seed),    %Previous customer arrived
      N == M+1,                     %Increment Id number
      T0 == T1 + poisson(Seed,NewSeed).%Delay for next customer
```

<u>Arithmetic</u>

The form of the arithmetic necessary for Starlog is worth a closer examination because time is a real number upon which arithmetic can be done. The difficulty here is that times are represented internally within a Starlog computation not just as single real numbers but also as intervals. The author in [Cleary, 1986] has examined the problem of correctly doing arithmetic in Prolog. The solution given there also used an underlying representation of real numbers as intervals. The resulting system had some interesting properties the most important for the current paper being that the resultant arithmetic behaved as a constraint solving system. For example an equation such as `X*X == Y` would remain dormant until one of X or Y was given a concrete value whereupon the value of the other would be computed. For example if later X = 2 then Y = 4 would be immediately computed. Or if it was known that X was in the interval [2,3] then Y would be computed to be in the interval [4,9]. Similarly if Y = 16 then then two solutions X = 4 and X = -4 will be computed. If Y = 2 then two small intervals X = (1.414,1.415) and X = (-1.415,-1.414) will be given as the solutions. These are the smallest intervals which surround the true solution of $\pm\sqrt{2}$.

The effectiveness of this can be seen in the following simple example:

```
p(T,X) ← q(T0), X == T**2, T < 100.
r(20,X) ← p(20,X).
```

If `q(10)` is placed into the pool then this rule is fired and the tuple `p([10,100),X)` is generated together with the attached constraint that `X == T**2`. No attempt is made to compute the (infinite) number of values of `X` in the interval, rather the system is lazy and waits until it needs to compute an actual value. For example, when `p` is matched with the body of the rule for `r`, `T = 20`, and at that point `X` is computed to be 400, so that, `r(20,400)` is placed into the pool.

This type of constrained arithmetic is particularly effective for combined continuous/discrete simulation problems. As an example we will consider the problem of a bouncing ball (for a different attempt at a solution to this problem see [Hale,1987]). In this problem a ball bounces from a floor, on each bounce losing some velocity. The trajectory of the ball is a continuous parabola between bounces but requires a single discrete event at each bounce. This process will be modeled by two tuples, `bounce` and `trajectory`. `bounce(T,V)` will occur at a single instant `T` and after the bounce the ball will have a vertical velocity `V`. `trajectory(T,Y,V)` specifies the vertical height `Y` and the vertical velocity `V` of the ball at time `T`.

```
1.    bounce(0,1) ←           %The starting bounce
2.    bounce(T,V) ←
            trajectory(T,0,W),      %The next bounce starts when the
      height is
                              %back to 0
            W < 0,            %and the velocity is downward
            V == -W * 0.5.    %Compute the new velocity

3.    trajectory(T1,Y,V) ←
            bounce(T0,V0), T == T1-T0,
            V == V0 - g*T, Y == V0*T - (g/2)*T**2,     %assume g = 1
            Y ≥ 0.
```

The table below shows the execution of this program through its first three bounces.

Tuple Pool (in time order) {constraints}	Number of rule used to generate tuple
`bounce(0,1)`	1.
`trajectory(T,X,Y){0≤T≤2, V==1-T, Y=T-(1/2)*T**2}`	3
`bounce(2,0.5)`	2
`trajectory(T,X,Y){2≤T≤3, V==2.5-T, Y=(T-2)-(1/2)*(T-2)**2}`	3.
`bounce(3,0.25)`	2
....	...

The only use that is made of the `trajectory` tuples above is to compute the next bounce. However, a typical task for this sort of problem would be to graph the resulting arcs. The following program sketches how this could be done. It assumes that the Starlog system has a graphical display and will blacken a pixel with integer co-ordinates `I,J` at the times when the tuple `pixel(T,I,J)` is true.

```
pixel(4,I,J) ←
        trajectory(T,Y,_),
        I+1 ≤ T*100, I ≥ T*100, is_integer(I,0,400),
        J+1 ≤ Y*100, J ≥ Y*100, is_integer(J,0,100).
```

The time the pixels are set is 4 which is the limit of the times when the bounces occur, this ensures that the clause is causal and the pixels are not set until after all the trajectory information is available. The co-ordinates are scaled by 100 and `is_integer(A,B,C)` is an arithmetic constraint that forces its argument `A` to be an integer between `B` and `C` `(B≤A≤C)`. The effect of all this is to find a solution for the `trajectory()` equations over the succession of time intervals [0,0.01], [0.01,0.02], ... [3.99,4.00] as generated by successive values for I. For example of I is constrained to be 0, then T will be constrained to the interval [0,0.01] and because of the equation from the first bounce,`Y=T-(1/2)*T**2`, Y is constrained to be [0,0.99995] and J to be 0. Thus the tuple `pixel(4,0,0)` will be generated blackening one dot on the screen.

This example shows how easy it is to print, or in this case graph, the answers generated by the system. This depends critically on two properties of Starlog: the ability to attach constraints to tuples; and the fact that all tuples in the system are public and accessible at all times.

Complex Bodies and Specialized Clauses

One situation we have not considered is how to deal with the execution of complex clauses where more than one tuple occurs in the body of the clause. Consider for example the following clause:

```
    p(T) ← q(T), r(T)
```

This says that the tuple `p(T)` will be generated whenever `q(T)` and `r(T)` occur at the same time. Imagine that the tuple `q(7)` is generated. Then one way to deal with this is to immediately generate two new specialized clauses. One has the reference to `q(T)` removed by bottom up resolution to form the new specialized clause:

```
    p(7) ← r(7)
```

This will only fire if the specific tuple `r(7)` is generated. To make this approach effective it is necessary to prevent the tuple `r(7)` also firing the original rule and thus leading to the new specialized rule `p(7) ← q(7)` and resultant duplicate computation The way to do this is to rely on the system generating new tuples in strict time order. So if `q(7)` has been generated then it is known that no other `q(T)` tuples will be generated at time 7 or earlier. Thus the original clause can be specialized to

```
    p(T) ← q(T), T>7, r(T).
```

So when `r(7)` does arrive it can fire only one of the two new rules. Thus the basic Starlog execution algorithm proceeds by continually specializing the set of existing clauses as new tuples tuples are generated. When finally a clause has been specialized so far that it has no body its head can be placed in the tuple pool as a newly generated tuple.

None of the examples so far have needed to consider this more complex form of execution because they have all contained only a single tuple call in their bodies. However, it will be important in understanding the colliding pucks example below.

A more complex example of the above process occurs when a tuple such as `q([8,9])` is generated. Then the two new specialized clauses which replace the original one are:

```
    p(T) ← r(T), T≥8, T≤9.
    p(T) ← q(T), T>8, r(T).
```

Now if `r(8)` is generated the resulting clauses are:

```
    p(8) ←
    p(T) ← r(T), T>8, T≤9. %Note change of condition to T>8 from T≥8
    p(T) ← q(T), T>8, r(T).
```

So a new tuple `r(8)` is generated and two clauses remain waiting to be fired.

Negation

The simple form of the language described above lacks a critical dimension of expressive power as it is unable to say that a tuple will be true for a period until some event occurs. This is critical for programs where it is necessary to change the state of a system on the occurrence of a particular event. In Starlog terms what is required is the ability to say that a condition is true so long as a particular tuple has not been placed in the tuple pool. A logical way to do this is to include negation in the set of allowable constructs in the body of a clause.

A simple example of the use of negation is the following program which mimics the effect of assignment of a state to an object. The state of the object at time `T` is represented by the tuple `state(T,V)` and the new state `W`, to which the objects value is to be set at time `T0`, is given by the tuple `new(T,W)`. Thus if the tuples `new(1,a)`, `new(3,b)` and `new(9,c)` occur then the tuples `state((1,3],a)`, `state((3,9],b)` and `state((9,∞],c)` should also be generated. They say that the state of the object from time 1 to 3 had the value a, from time 3 to 9 the value b, and thereafter the value c. The following single clause program accomplishes this:

```
1a    state(T,V) ←
1b          new(T0,V), T ≥ T0,
1c          not(exists T1,W new(T1,W), T1>T0, T>T1).
```

Line 1b of this clause says that `state(T,V)` will be true at time `T` if `new(T0,V)` has occurred at some earlier time T0. If this were all that was in the clause then the object would have multiple possible values. For example at time 4 it would have both the values `a` and `b` and at time 10 the three values `a`, `b` and `c`. Line 1c prevents this by saying that a state tuple is true only until another `new()` tuple appears. To paraphrase the negation it says that `state(T,V)` will be false if there exists some tuple `new(T1,W)` whose time is after `T0` and before `T`. So after the tuple `new(1,a)` appears it will match the clause and generate a specialized instance:

```
1a    state(T,a) ←
1b          T ≥ 1,
1c          not(exists T1,W new(T1,W), T1>1, T>T1).
```

That is the tuple `state(T,a)` will be generated with the constraint that `T≥1` and with the additional constraint implied by the negation. The tuple `new(1,a)` does not match `new(T1,W)` in the negation because of the constraint that `T1>1`. However, when the tuple `new(3,b)` appears it does match and reduces the clause to:

```
1a    state(T,a) ←
1b          T ≥ 1,
1c          not(~~3>1~~, T>3).
```
(~~3>1~~ has been struck through because it is trivially true). Reduced to interval notation this gives the tuple `state((1,3],a)`.

Semantics

Starlog without negation has a very simple least fixed point semantics essentially identical to pure Prolog (the only differences is the introduction of real numbers). However, the introduction of negation makes the semantics much more problematic. In pure Prolog negation can be dealt with in stratified programs. These are ones where it is possible to layer or stratify the program in such a way that predicate at a particular layer depends only on predicates at the same or lower layers and if negation is used that the dependency be to predicates strictly lower in the stratification.

A similar stratification is used in Starlog except that now the stratification is in time. The basic idea is that the tuple that depends on a negation can only refer back to tuples that occurred strictly in the past. In fact the precise condition is that there cannot be a zero delay loop around a cycle of calls if they involve negation. The upshot of this is that any program that is temporally stratified in this way has a well defined and unique minimal model.

Escape to Prolog

It is convenient to be able to write routines such as `is_integer()` directly. It is difficult to do this in Starlog itself. However, standard Prolog is well suited to the task. Accordingly Starlog includes an escape mechanism to Prolog. For example, `is_integer()` could be written in Prolog as follows:

```
    is_integer(B,B,C):- B≤C..
    is_integer(A,B,C):- An == A+1, An≤C,. is_integer(An,B,C).
```
Given values for B and C it will successively generate each integer value from B to C. Some control is needed over when this will be invoked from the Starlog level, for example, it will generate an infinite number of not very useful solutions if the values of B and C are uninstantiated. Borrowing from NU-Prolog [Thom and Zoebel,1988] the following notation specifies that `is_integer(A,B,C)` should only be called when all of `A`, `B` and `C` have values:

```
is_integer(A,B,C) when A and B and C.
```

Multiple Heads

One additional construct is very helpful when writing programs such as the colliding pucks example below. This allows a clause to have more than one head. For example:

```
p(T),q(T) ← r(T)
```

says that when the rule is fired by the arrival of the tuple `r(T)` then both of `p(T)` and `q(T)` are generated as new tuples. For example when `r(42)` occurs `p(42)` and `q(42)` are both generated. This extension is not a fundamental one as the single clause above can be treated as an abbreviation for the two clauses:

```
p(T) ← r(T)
q(T) ← r(T)
```

In standard Prolog such an abbreviation is not useful but the rather different coding style of Starlog makes it very helpful.

Colliding Pucks

As an illustration of how Starlog can be used for a non-trivial distributed program I will now develop an example program to solve the colliding pucks problem. This problem has been used in a number of benchmark studies of distributed simulations. In its simplest form it involves a frictionless billiard table on which are placed a number of pucks. These pucks are all moving and bounce off one another and off the four walls of the billiard table. The first example below gives a simple sequential solution to this problem and uses negation as a critical part of its coding.

Each puck is modeled by two classes of tuples. The tuple `collide(T,N,P,V)` says that at time `T` a puck `N` collided (either with a wall or another puck) and started on a new trajectory. `P` gives the (vector) position of the center of the puck at the start of the trajectory, and `V` the vector velocity of the puck at the start of the trajectory. The tuple `trajectory(T,N,V,X)` says that at time `T` the puck `N` will be at vector position `X` as a result of the initial velocity at the beginning of the trajectory `V`. `N` is an integer uniquely identifying the puck. (It will be assumed that all pucks have the same unit radius and mass).

`trajectory` is defined by saying that it is the sequence of positions of the center of the puck until the next collision occurs. These are specified by using a constraining equation as in the bouncing ball example. The condition that the trajectory ends at the next collision is applied by using negation. The clause specifying `trajectory` is:

```
     trajectory(T,N,V,X) ←
            collide(T0,N,P,V), T≥T0,
            X == P + (T-T0)*V, % assume vector arithmetic is provided
            not(exists T1,P',V' collide(T1,N,P',V'), T1>T0, T>T1).
```

This is almost identical to the clause for `state()` given in the example above. There is one key difference however, the identifier `N` which occurs in `collide(T0,N,P,V)` is used inside the negation to select just the collisions which involve that puck. Otherwise, the tuple would only be true until the next collision of <u>any</u> puck, not just the one currently under consideration.

When solving this problem in more conventional languages the usual technique is to compare all pairs of pucks and their trajectories. It is also necessary to keep a record for each puck of the next collision it will undergo. It is possible that as the comparison proceeds that a new collision will be found that is earlier in time. The result is that the previous putative collision must be removed and the other puck that was to participate in the collision must be reconsidered for alternative future collisions. This algorithm is surprisingly complex for such a simple problem. This difficulty is automatically dealt with, however, in the Starlog program above which does not have to explicitly keep track of future collisions for each puck. Rather it is taken care of by the negation call which limits a trajectory to the time up to the next collision.

The `collide` tuple is generated when two pucks on a trajectory touch (their centers are separated by their radii). Each such collision involves two pucks coming into the collision and the same two outgoing. The clause below thus uses the multiple head abbreviation to generate two outgoing `collide()` tuples:

```
     collide(T,N1,X1,V1new),
     collide(T,N2,X2,V2new) ←
            trajectory(T,N1,V1,X1),
            trajectory(T,N2,V2,X2),
            N1 < N2, %break the symmetry of the situation and prevent a
                   %collision from occurring twice
            physics(V1,X1,V2,X2,V1new,V2new).
```

The clause says there will be a collision at time `T` if the instances of the trajectories at time `T` satisfy the conditions given by the Prolog routine `physics()`. Note that the positions for the collisions are given by the positions of the pucks trajectories at time `T`. New velocities are generated as a result of the collision. To prevent the same collision from being computed twice with the `trajectory()` tuples in the opposite order the arbitrary constraint that `N1<N2` is imposed.

The only reason to code `physics()` as a separate routine is to isolate the physical aspects of the problem It is given a null when statement so that it will be expanded in-line. That is, when the Starlog system first starts, it will generate a single more complex clause by eliminating the call to `physics()`.

`physics()` is an interesting routine which shows the power of the constraint based arithmetic used here. The code assumes that vector arithmetic is provided by the system, and that the dot product of two vectors is given by the operator •.

```
        physics(V1,X1,V2,X2,V1new,V2new) when ever.
        physics(V1,X1,V2,X2,V1new,V2new):-
1               D == X1-X2, D•D == 1,    %the  centers are one radius apart
2               V1new+V2new == V1+V2,    %Conservation of momentum
3               R == D*[[0,-1],[1,0]],   %Rotate D by 90 degrees
4               V1new•R == V1•R,         %Velocities perpendicular to the
        centers
5               V2new•R == V1•R.         %are unchanged
```

Line 1 computes the vector `D` which lies between the centers of the pucks. The square of its length is specified to be 1 which corresponds to the two pucks having their edges touching. Line 2 specifies that the momentum of the outgoing pucks is equal to that of the incoming ones. Line 3 computes `R` to be `D` rotated by 90 degrees (using a matrix multiplication). `R` is then used in lines 4 and 5 to specify the constraint that the velocities perpendicular to the centers remain unchanged before and after the collision. Because the arithmetic is constraint based it is not necessary to rearrange it to explicitly compute the values of the outgoing parameters. The system is smart enough to propagate the constraints and solve for these values.

To initialize the problem with some pucks a number of `collide()` tuples should be specified at time 0 with the initial velocities and positions of the pucks. An example might be:

```
        collide(0,1,[1,1],[0.5,0.4]) ←
```

Care is needed that all the initial tuples are specified at time 0 and that all the puck identifiers are different integers.

To complete the problem it is necessary to include collisions with the walls. This can be done by adding a single clause which generates `collide()` tuples.

```
collide(T,N,X,Vnew) ←
      trajectory(T,N,V,X),
      wall(X,M),
      Vnew == V*M.
```

The Prolog routine `wall ()` is given below. It specifies the four walls of the billiard table. The first parameter is a constrained equation specifying the location of the center of a puck when it is just touching a wall. The second parameter is a matrix which says how the velocity is transformed when the collision occurs.

```
wall(X) when ever.
wall([X,Y],[[1,0],[0,-1]]):- X=0.5.
wall([X,Y],[[1,0],[0,-1]]):- X=S-0.5, size(S).
wall([X,Y],[[-1,0],[0,1]]):- Y=0.5.
wall([X,Y],[[-1,0],[0,1]]):- Y=S-0.5, size(S).

%The size of billiard table (assumed to be square).
size(10).
```

`wall()` is specified to be expanded immediately by its `when` specification, in the same way as the routine `physics()`. The result is that four new clauses are generated at the start of execution, one for each clause of `wall`. Each of these has its own constraining equations on the positions and velocities.

This program is clearly O(n^2) in n the number of pucks. As each trajectory must be compared against every other trajectory in existence at the same time. The n^2 computation occurs in the first clause for `collide()`. The two occurrences of trajectory in that clause mean that as a tuple is generated it will be matched against each of the trajectory calls giving two new specialized clauses, one with the first call eliminated and the other with the second call eliminated (see previous section for a more detailed description of this process). Each of these O(n) new clauses will be available to receive a tuple in the remaining call. Each will be restricted to the time range of the appropriate tuple so the ignominy is avoided of having to compare against all tuples over all time making an O(n^2t^2) computation (where t is length of time of the simulation). An O(n^2) computation is to be expected for any similar algorithm and is not a function of the language that we have used. Any improvement in this computation time must come from an improvement in the algorithm. Incidentally, this algorithm could possibly be speeded up by using a multicomputer. However, it is much more fruitful to consider the more efficient algorithm of the next section and its very efficient distributed implementation.

Sectorized Colliding Pucks

Basic Algorithm

One way to improve the algorithm above is to divide the billiard table into small square sectors. Then when checking a puck for collisions only those pucks lying in the same sector or in immediately neighboring sectors need be checked for potential collisions. (This makes an assumption that the sectors are all larger than the radius of the largest puck). To do this we will break up each trajectory into a series of segments, each segment corresponding to the portion of a trajectory within one sector. To do that all that we need do is introduce a dummy `collide` event at each sector boundary. This is accomplished with a single new clause added to our previous program:

```
collide(T,N,X,V) ←
      trajectory(T,N,V,X),
      boundary(X).
```

The Prolog routine `boundary()` specifies the set of lines along the boundaries of all the sectors.

```
boundary(X) when ever.
boundary(X) :-
      size(N),          %Sectors are 1x1 squares
      is_integer(I,0,N),      %Generate I = 0, 1, 2, ... N
      s(I,X).           %Set equations for each boundary

s(I,[X,Y]):- X=I. %The horizontal boundaries
s(I,[X,Y]):- Y=I. %The vertical boundaries
```

Because the `boundary()` call is expanded immediately this leads to a number of new clauses, one for each boundary line on the table.

Sector Indexing

Unfortunately while this gives us the correct logic for the new algorithm it does not help speed the computation. The problem is that while we have carefully subdivided the trajectories each clause still accepts all the trajectory tuples for comparison. Thus there is nothing in the clauses to prevent pairing trajectories from distant parts of the table. In fact all that has been done is to increase the number of trajectory tuples and the total amount of computation. What is needed is some way to specialize the clauses in such a way that a given specialized clause will only accept tuples generated by a small number of other specialized clauses. To see a simple example of this consider the following example program:

```
1       p(T)← q(T,1)
2       p(T)← q(T,2)

3       q(T,1)← ...
4       q(T,2)← ...
```

The two clauses for `p` will by default try and compare all incoming `q()` tuples. However, clause 1 will only ever match against tuples generated by clause 3 and similarly for clauses 2 and 4. This is an example of the *indexing* problem which is well known in standard Prolog implementations. The solution which is straightforward to implement in Starlog is to restrict the comparison of incoming tuples in clauses 1 and 2 just to the appropriate tuples generated by clauses 3 and 4.. This will halve the amount of work being done to fire the clauses.

This can be carried over to the colliding pucks problem by using the names for the different sectors on the table to force indexing of the clauses. The idea is to use Prolog clauses to generate a large number of alternative specialized clauses at the beginning of the computation. Each of these clauses would deal with collisions and trajectories in just one sector and by the use of indexing would receive input only from the appropriate specialized clauses in neighboring sectors.

To do this each of the `collide()` and `trajectory()` tuples will be given an extra final parameter corresponding to the name of the sector in which they occur. The sectors will be named by the (integer) co-ordinates of their top right hand corners. The first step in the new version of the program is to define a Prolog routine `sector(S,X)` which specifies the set of points in each sector:

```
        sector(S,X) when ever.
        sector([I,J],[X,Y]):- range(X,I), range(Y,J).

        range(V,K):- size(N), is_integer(K,1,N), V≤K, V≥K-1.
```

The first parameter, `S`, is the name of the sector and the second parameter, `X`, is a vector which will be constrained to be in the sector. The routine `range(V,K)` generates `K` as all possible integers from 1 to N (the number of sectors on the table - for simplicity it has been assumed that the table is NxN).

I will now revisit each of the `trajectory` and `collide` tuples and compute the sector they should lie in.

```
collide(T,N,X,V,S1) ←
        trajectory(T,N,V,X,S), sector(S,X), sector(X,S1), S≠S1,
        boundary(X).
```

This clause is the one to compute when a boundary pseudo-collision occurs. The outgoing tuple is assigned to the sector over the boundary from the sector which originated the trajectory. This is done by noting that every boundary point belongs to two sectors. When boundary and sector are both expanded during initial computation their constraints on X will interact so that each sector will get 4 specialized clauses generated, one for each boundary of the sector. After specialization and indexing the clauses will receive only `trajectory()` tuples from within their own sectors.

```
collide(T,N,X,Vnew,S) ←
        trajectory(T,N,V,X,S), sector(S,X),
        wall(X,M),
        Vnew == V*M.
```

This clause computes the collisions against the walls. The collision occurs in the same sector as the trajectory. In this case there will be one specialized clause generated for each sector which shares an edge with a wall. After specialization and indexing these clauses will receive `trajectory()` tuples only from their own sectors.

```
collide(T,N1,X1,V1new,S1),
collide(T,N2,X2,V2new,S2) ←
        trajectory(T,N1,V1,X1,S1), sector(S1,X1),
        trajectory(T,N2,V2,X2,S2), sector(S2,X2),
        N1 < N2, %break the symmetry of the situation and prevent a
                %collision from occurring twice
        physics(V1,X1,V2,X2,V1new,V2new).
```

This clause is the one that computes the collisions between pucks. The modifications to the first two calls in the body ensure that the sector of a collision is the same as that of the appropriate incoming trajectory. Naively, this should generate N^2 specialized clauses one for each possible pair of sectors. However, the selection of a sector S1 or S2 constrains the values of X1 and X2. But X1 and X2 are also constrained by the equations of `physics()` to be within a distance 1 of one another. It turns out that there is just enough information for the system to deduce that there is no possible solution for sectors which are not neighbors. So most of the N^2 specialized clauses will fail leaving 9N specialized clauses. After indexing each of these specialized clauses will receive tuples from just two sectors (one for each of the two `trajectory()` calls). This is what was sought and reduces the computation in the algorithm substantially.

```
trajectory(T,N,V,X,S) ←
      collide(T0,N,P,V,S), T≥T0,
      X == P + (T-T0)*V, % assume vector arithmetic is provided
      not(exists T1,P',V',S'
            collide(T1,N,P',V',S'),
            neighbor(S,S'),
            T1>T0, T>T1).
```

This clause is the one that defines trajectories from a particular collision. Clearly the `trajectory()` lies in the same sector as the initiating `collide()` tuple. After specialization and indexing there will be one specialized clause for each sector and it will receive `collide()` tuples only from the `collide()` clauses specialized to the same sector. However, the negation also receives `collide()` tuples and without further care they will be accepted from all the specialized clauses in the system. It can be seen that the next `collide()` tuple in the negation must be generated in either the same or a neighboring sector. This condition is added by the Prolog routine `neighbor()` which specifies that two sectors are neighbors if they have at least one point in common.

```
      neighbor(S1,S2):- exists X ( sector(S1,X), sector(S2,X).
```

After specialization and indexing this will lead to a slightly more complex form for the negation which will receive `collide()` tuples only from neighbors.

Distributing Computations

Mapping

The previous section has specified a relatively efficient algorithm for simulating colliding pucks. However, the algorithm is still a sequential one for no mechanism has been given for distributing it over a multicomputer system. One way to do this is to position the clauses in the system so that one or more clauses are run on each processor. This makes most sense when the units being positioned are specialized clauses. The greater number and restricted connectivity of such clauses makes them ideal candidates for positioning and for optimizing the communication streams between processors. For example, in the colliding pucks example above if all the clauses associated with a sector are placed on one processor then newly generated tuples need be sent only to those processors which are dealing with neighboring sectors and not broadcast to all processors.

To do this Starlog allows mapping declarations to be added to programs. These have the following form:

```
map <tuple> to <processor> where <condition>.
```

For example,

```
map collide(T,N,V,X,S) to Proc where mapper(S,Proc).
```

`<tuple>` refers to the tuple which occurs in the head of a clause which is to be mapped. `Proc` refers to some processor name which has external significance outside the program and which specifies a unique processor. `<condition>` is a Prolog routine which can be used to associate the `<processor>` with particular parameters in the tuple. For example, if the sectors above are to be mapped to 1 of Np processors named 0 thru Np-1 then the following code might be used for the mapping:

```
mapper([I,J],Proc):-
      number_procs(Np), size(N),
      Proc == ((I-1) + (J-1)*N) mod Np.
```

In some cases it will be ambiguous where a clause should be positioned. Either because the clause is insufficiently specialized to provide a unique processor number, for example, if the sector was uninstantiated then the clause could potentially be mapped anywhere on the system. Another possible reason is multiple heads in a clause, as for example in the clause which computes the collisions between pucks. The easiest way to resolve this is to allow the system to make a free choice when there is more than one possible mapping.

Optimistic Computation

Once the individual clauses have been placed on processors it is still necessary to specify a synchronizing mechanism between the different processors. In Starlog programs which do not contain negations this is very simple. Such programs are monotonic, that is the generation of a tuple can only cause new tuples to be generated, but can never cause them to be withdrawn. Because of this each processor can compute freely accepting incoming tuples from other processors as they come and firing rules which match them.

However, when negations are present in a clause it is necessary to know that a tuple cannot arrive before generating the outgoing tuple in the head. In a non-distributed implementation it is possible to examine all possible tuples which could match the negation and find the earliest possible time at which the negation could occur. If this is after the current time then the tuple can safely be generated in the knowledge that the negation can never be fulfilled. In a distributed system it may be that a tuple is generated on a remote machine which fulfills the negation but which arrives late. One way to deal with this is to compute optimistically by assuming that unless evidence to the contrary is received no tuples negating a clause will be received. The tuples generated as a consequence (and their descendants) may have to be removed later when a negating tuple arrives. One simple mechanism for administering this is to is to attach the negation to a tuple as a form of constraint. If later a negating tuple arrives all those tuples which have inherited the constraint will immediately fail. The constraints can be eventually garbage collected as Global Virtual Time advances and it is known that no possible negating tuples could arrive.

Acknowledgements

This work was partially funded by the Natural Sciences and Engineering Research Council of Canada.

References

Carriero, N.; and D. Gelernter 1989. "LINDA in Context." *Comm. A.C.M.* 32, no. 4 (April): 444-458.

Cleary, J.G. 1987. "Logical Arithmetic." *Future Computing Systems* 2, no. 2: 125-149.

Galton, A. Ed. 1987. *Temporal Logics and their Applications*. Academic Press, London.

Hale, R. 1987. "Temporal Logic Programming." in *Temporal Logics and their Applications* (A. Galton, ed.). Academic Press, London, 91-119.

Jefferson, D. 1985. "Virtual Time." *Trans. Programming Languages and Systems* 7, no. 3 (July): 404-425.

Lloyd, J.W. 1987. *Foundations of Logic Programming*. Springer-Verlag.

Thom, J.A. and J. Zobel 1988. "NU-Prolog Reference Manual- Version 1.3." Technical Report 86/10. Department of Computer Science, University of Melbourne..